

Building OpenOffice.org with -j50 and beyond

August 25, 2006 kaib@google.com

Summary

OpenOffice.org is a 5 million line project that needs to be built simultaneously on several platforms (windows, linux, solaris, mac) using different configurations (production, debug). This paper presents a way to aggressively distribute this task to a heterogeneous shared cluster of build machines running the desired target build environments and having common storage. The developer workstation is running jam to control and correctly handle a mix of platform specific and common targets.

Overview

1. Generate single full project dependency graph, including proper dependencies for all target files on each platform we are building.
2. Set the PLATFORM target specific variable on each output file to indicate what platform it needs to be built on.
3. Set JAMSHELL to a site specific utility that runs an arbitrary shell command on one of the build cluster machines that has the correct platform type. The output goes into the shared storage.
4. Let jam figure out the action order and execute the update actions.

Building the dependency graph

The first step is building the complete dependency graph of all targets and source files into memory. To the developer all this is transparent, here is a sample Jamfile from a source directory:

```
SubDir TOP sc source ui app ;

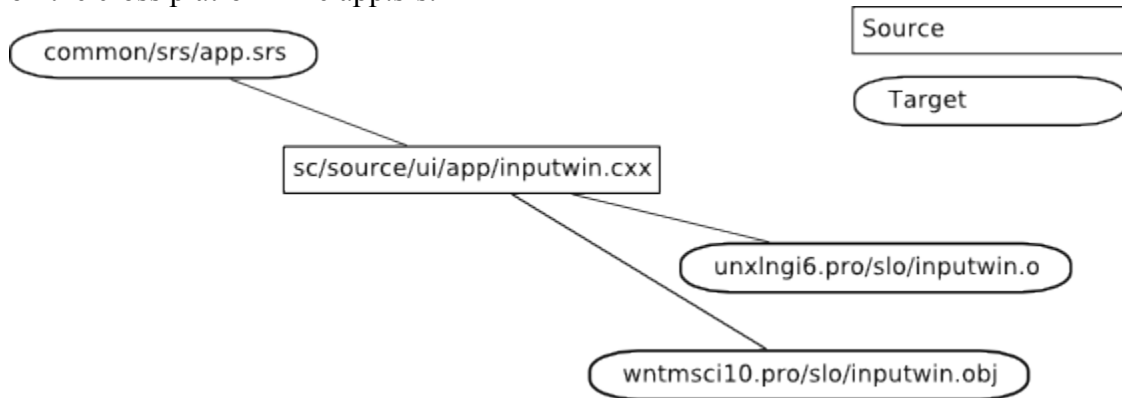
# Define source files
CXXFILES on sc-app = [ Glob $(SEARCH_SOURCE) : *.cxx ] ;

# Compile and link
CxxSourceDirCompile sc app ;
CxxSourceDirLib sc app : app ;
```

The target platforms are found in the variable TARGET_PLATFORMS which can be imported from the environment, a common configuration file or overridden by the developer himself. There is a similar variable TARGET_CONFIGURATIONS enumerating the configuration (production, debug) we desire to build. We'll assume for now that:

```
TARGET_PLATFORMS = wntmsci10 unxlngi6 ;
TARGET_CONFIGURATIONS = pro ;
```

Here is a piece of the dependency graph around the file `inputwin.cxx`, that also depends on the cross platform file `app.srs`:



Note how our jam rules have created two platform specific object files and only a single common srs target file. This complete dependency graph guarantees we update all required targets once and only once.

Setting the PLATFORM variable on targets

Jam supports a mechanism of target specific variables:

```
GREETING = "Hello world!" ;
GREETING on mytarget = "Hi everyone!" ;
```

```
MyAction generictarget ;
MyAction mytarget ;
```

```
actions MyAction
{
    echo $(GREETING)
}
```

```
This will output:
MyAction generictarget
Hello world!
MyAction mytarget
Hi everyone!
```

The build process will use this feature to set the following PLATFORM values:

```
PLATFORM on common.pro/srs/app.srs = common ;
PLATFORM on unxlngi6.pro/slo/inputwin.o = unxlngi6 ;
PLATFORM on wntmsci10.pro/slo/inputwin.obj = wntmsci10 ;
```

Running update actions and distexec

Once the dependency graph has been built jam starts running update actions in parallel. Here is a sample update action for compiling C++ files:

```
actions CxxCompile
{
    $(PLATFORM) $(CXX) $(1) $(2)
}
```

We have earlier set the variable JAMSHELL to our custom shell distexec:

```
distexec <job_num> <platform> <shell_command>+
```

The task of distexec is simple: find a suitable machine in the build cluster and execute the shell commands on it. In case <platform> is “common” any machine will do, otherwise we need to pick a machine with the correct platform. This could be a simple shell script using fsh or rexec, or a fast C app doing load balancing among the machines. The command might also need to re-root the paths to the shared storage and convert them to native format on the platform.

Questions

What is run where in the distributed scenario?

The build is controlled by the developer running jam on her workstation. Actions are shipped out the build cluster machines which only need to have the necessary platform tools and access to the storage. All source files and target files need to reside on the common storage.

Can I build on a local disk without penalty? And still use my multi CPU/core?

Yes and yes. You can configure the process not to set JAMSHELL and the PLATFORM variable. You will end up with a local disk build that can still benefit from dual core or multi CPU machines.

Will compile batching and precompiled headers work?

Yes, as long as your compilers on the machines compiling a single platform are compatible. The pch files will be generated for each platform separately and then shared by the cluster machines via the shared storage. Batching works as usual.

Ccache? Icecream/distcc?

You might be able to set up an environment to use ccache and get some speedup out of it. You just need to make sure you share the ccache state between the build machines. Icecream and distcc duplicate the functionality presented here and don't support precompiled headers so they probably aren't worthwhile. If you compile on a single machine you can use them normally.

Why Jam?

You don't need jam but it has everything out of the box. What you really need is a tool that can hold the dependency graph in memory and keep track of the platform each target needs to be built on. In most make tools you might be able to generate variable names programmatically and use this mechanism to store the platform for target files.

The bigger problem is that GNU make and similar tools only know a single type of dependency. This means each object file ends up depending on every file included directly or indirectly giving a total memory consumption of $O(n^2)$ where n = the total number of source files. Because each platform-configuration combination needs it's own object file you will end with a total memory consumption of $O(mn^2)$. For a project like OO.o this means GB's RAM to hold the dependencies. Even if you had enough memory you would still need to read all the dependency data from disk with a huge startup penalty.

Currently SCons supports a similar *include* style dependency as jam. The memory consumption for both is $O(m + n)$.